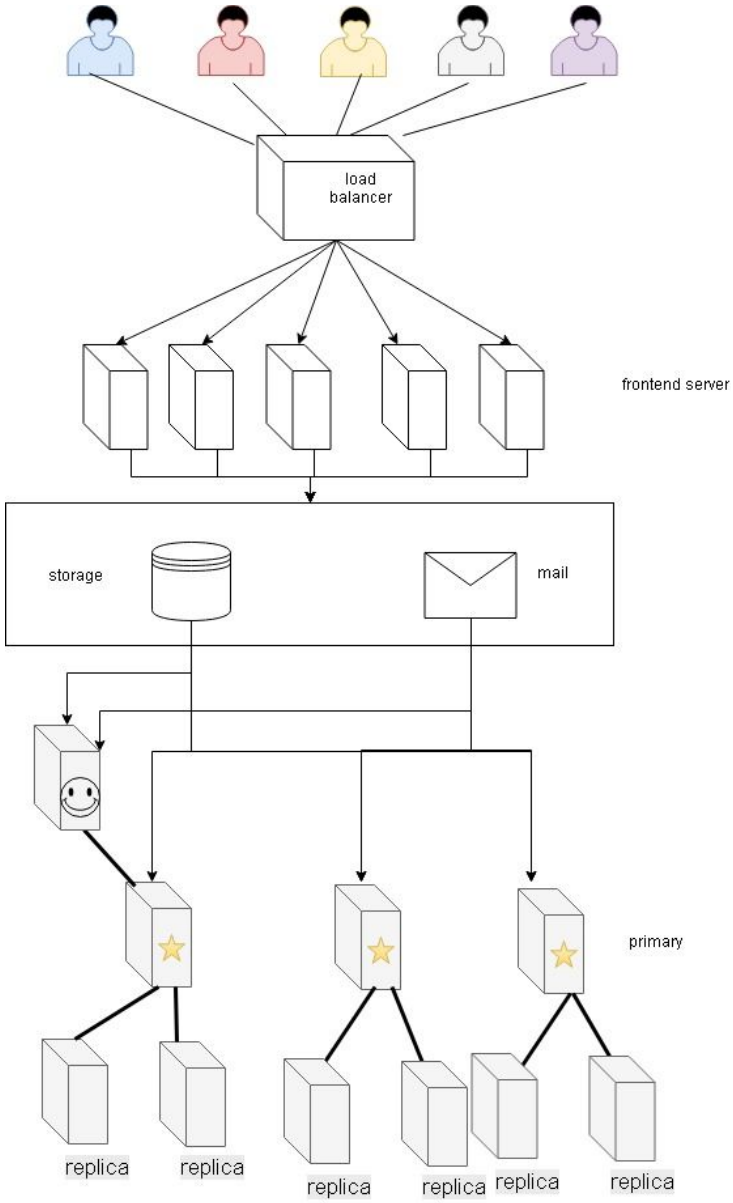
CIS 505 Project Report
--PennCloud

ruqing@seas.upenn.edu Ruqing Xu
xiangli3@seas.upenn.edu Xiang Li
liuya@seas.upenn.edu Yang Liu
wudao@seas.upenn.edu Wudao ling

1. Architecture Diagram



2. Features

a. Webmail

The webmail storage design is that each user has 2 email lists in backend, 1 for inbox and 1 for sent mail. These lists record MD5 hashes of emails, and hash can be used to localize email in key-value pair, for example, GET <user> <mail/inbox/hash>. CPUT is used to modify hash lists for consistency. After CPUT successes, it will be safe to PUT or DELETE email.

Webmail services consists of sending, extraction and update. Mail sending strictly follows SMTP protocol. One of frontend servers will parse information including sender, recipient and content from html, then call a SMTP client to communicate with a SMTP server. When SMTP server receives an email to localhost mailboxes, it will generate a hash based on email content, use GET and CPUT to append hash lists (sender's sent list and recipient's inbox list) and then use PUT to add emails in backend. When the email has an outside recipient, SMTP server will save them in a local message queue. Later a mail deliver will use DNS to look up domain name, choose the host with highest priority, send mail to that SMTP server and update message queue.

Different from sending, extraction and update are directly applied to key-value store. Extraction simply use hash lists to read a vector of emails and return them to frontend for display. Update takes username, email hash and email type (in or out) from frontend and delete this email in backend, which is CPUT hash list then DELETE email.

Besides, we also support reply and forward function. Basically they are still sending function, but frontend server will pre-fill some information such as email subject, reply recipient and email content.

b. Storage

User is able to upload/delete file, create/delete folder, rename file/folder, move file/folder, back to any directory.

When a user logs in, it will create a Storage object to handle the command from user. The storage will also keep track of the user's current directory.

We use the created time of a file/folder as the id of that file/folder, the content of a certain folder ID in the kvstore stores all the file/folders type, id, name, size, time in that folder path. The content of a certain file ID in kvstore stores the content of that file(as a hex string).

To handle the case that two people log into the same user account, we send cput command to backend instead of put when we want to make some change in key-value store. If two people try to do the cput at the same time, if one of them succeeds, the the content of that cell after cput is different from the data get by the other server, then the other server's cput will definitely fail.

Also, we handle the case that the primary server crashes, if the primary server crashes, storage will connect to master server again and ask for the new address of the current primary and then connect to that primary server.

c. Frontend

We use two classes request and response to handle HTTP get and post request.

d. Cookie

The username is used as a cookie to distinguish different users. The user will keep logged in until the browser is closed or the user choose to log out.

3. System Design

a. Consistency Model

Remote write - only the current primary node can accept a PUT/CPUT/DELETE request. When a primary node receives a request, it will generate a sequence number and pushes the update to all the replicas with the sequence number to achieve a virtual synchrony . On success, the primary will reply a SUCCESS message to the client.

Like GFS, if a client receives a SUCCESS message, it is believed that the update have been taken place in primary and all the replicas. Otherwise, the client should retry the command again. Due to the nature of key-value store, duplicate commands should not affect the correctness of the data.

b. Fault Tolerance

i. Heartbeat

Every primary node sends a heartbeat message to master server every second. When a primary node fails, master server should detect it at once. The master server will try to contact an alive replica and promotes it to the new primary. It will broadcast the decision to all the replicas.

ii. Recovery

Each data server keeps a log - when a command is received, it will force-append the command with the sequence number to the local log. Also it keeps a periodic checkpoint with a version number on local disk.

When a data server tries to recovery from crash fail, it will :

1. Try to contact master server
2. Ask master about the new primary
3. Try to contact the new primary server
4. Look at local log, find the latest sequence number
5. Send recover request to primary (with the latest log number)
6. If the reply is +OK, it will replay local checkpoint and log. Otherwise, it will request log and checkpoint from primary and replay the checkpoint and the log
7. Report "Ready" to primary node, waiting for updates from primary's holdback queue

c. Load Balancer

We use a special url http://localhost:8090 as a load balancer, the user only need to access to this url and the load balancer will redirect the user to one of the frontend servers . The strategy for the load balancer is similar to least recently used. We use a deck to store all the servers and check the state of each server every 5 seconds and always use the server at the front of the deck and if the server is down or the it has handled a new request we will move the server to the back of the deck. If all the servers are down, an error message will be returned to the user.

4. Division of Labor

- Key-value backend server: Ruqing Xu
- Storage server: Xiang Li
- frontend server: Yang Liu
- Mail server: Wudao Ling